Búsqueda binaria: aprende a implementar en Python

B brendomatos
20/06/2022

Ya entendemos la importancia del [análisis de complejidad de los algoritmos](#) para el desarrollo de software con un rendimiento cada vez mejor. ¿Así que, continuamos con nuestro aprendizaje?

En el artículo mencionado anteriormente, presentamos un problema de búsqueda que resolvimos a través del algoritmo de búsqueda binaria. Al final, dejamos un reto para resolver: ¿qué tal si lo implementamos en Python?

Desarrollaremos una búsqueda de estudiantes para Alura Latam. Después de encontrar a la persona, nuestra intención es autenticarla para autorizar el acceso a la plataforma. Para ello recibimos un listado con todos los nombres de las personas registradas.

En primer lugar, usaremos la búsqueda lineal.

Entonces decidimos revisar toda la lista buscando a la persona a autenticar y llegamos a la siguiente implementación:

```
from array import array

def buscar(lista, nombre_buscado):
    tamano_lista = len(lista)
    for actual in range(0, tamano_lista):
        if (lista[actual] == nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos = sorted(importa_lista('../data/lista_alumnos'))
    posicion_del_alumno = buscar(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la posicion {}".format(lista_de_alumnos[posici

if __name__ == "__main__":
    main()
```

¡Perfecto! Desarrollamos la función de búsqueda con una lista que contiene 7 estudiantes y también simulamos la búsqueda con aproximadamente 85,000, el número aproximado de estudiantes en Alura. Todo funcionó bien y con un rendimiento aceptable, como muestra el siguiente algoritmo:

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(archivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista
```

```
def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0, tamano_de_lista):
        if (lista[actual] == nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos = sorted(importa_lista('../data/lista_alunos'))
    posicao_do_aluno = busca(lista_de_alumnos, "Wanessa")
    print("Alumno(a) {} está en la posicion {}".format(lista_de_alumnos[posici

if __name__ == "__main__":
    main()
```

¡El gran problema!

Todo va sobre ruedas, pero supongamos que los responsables del equipo comercial nos contactaron informándonos que cerramos nuevas alianzas con grandes empresas. Por lo tanto, recibiremos aproximadamente 3000 autenticaciones de personas nuevas al día siguiente.

Sintiéndonos preocupados por la calidad de nuestro servicio, decidimos hacer una simulación simple para ver cómo reaccionaría nuestra búsqueda ante esta cantidad de solicitudes. Y pensando en el peor de los casos, siempre optaremos por la búsqueda de la última persona de la lista.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
```

```
    for line in lines:
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    for actual in range(0, tamano_de_lista):
        if (lista[actual] == nombre_buscado):
            return actual

    return -1

def main():
    lista_de_alumnos = ["Brendo", "Erica", "Monica", "Nico", "Paulo", "Rodrigo
    for i in range(0, 3500):
        posicion_del_alumno = buscar(lista_de_alumnos, "Wanessa")
        print("Alumno(a) {} está en la posicion {}".format(lista_de_alumnos[po

if __name__ == "__main__":
    main()
```

Cambiamos la llamada de búsqueda y recorrimos un bucle para simular fácilmente las 3500 solicitudes de búsqueda. Al correr notamos que tomaba un tiempo considerable.

Pero ¿por qué sucedió esto?

Hagamos un cálculo simple para responder a esta pregunta:

Para cada una de las 3500 solicitudes de búsqueda se realizaron 85 mil comparaciones, es decir: (3500*85000) = 297500000 operaciones.

Y si la lista de personas o solicitudes crece, el algoritmo crecerá linealmente a estas cantidades. Así que asumimos que cada uno de estos algoritmos es O(N) (hablaremos más sobre la notación Big O más adelante en este artículo).

Ahora vamos a optimizar el algoritmo.

Pensando en los alumnos que recibiremos, vamos a optimizar este algoritmo utilizando la técnica de divide y vencerás mediante búsqueda binaria.

```
from array import array

def importar_lista(archivo):
    lista = []
    with open(arquivo) as tf:
        lines = tf.read().split('","')
    for line in lines:
        lista.append(line)
    return lista

def buscar(lista, nombre_buscado):
    tamano_de_lista = len(lista)
    inicio = 0
    fim=tamano_de_lista-1

    while inicio<=fin:
        medio=(inicio+fin)//2
        if lista[medio] == nombre_buscado:
            return medio
        elif lista[meio] < nombre_buscado:
            inicio=medio+1
        elif lista[medio] > nombre_buscado:
            fin = medio-1

    return -1

def main():
    lista_de_alumnos = sorted(importar_lista('../data/lista_alumnos'))
    for i in range(0,3500):
        posicion_del_alumno = buscar(lista_de_alumnos, "Zeina")
        print("Aluno(a) {} está en la posicion {}".format(lista_de_alumnos[pos
```

```
if __name__ == "__main__":
    main()
```

En primer lugar, usamos la función `sorted()` de Python para devolver la lista en **orden alfabético**. A partir de ahí, empezamos a buscar a la alumna Zeina, que está al final de la lista, para simular el peor de los casos.

Refactorizamos la función de `buscar` para utilizar la búsqueda binaria, que consiste en comparar el valor buscado con el valor del elemento en el medio de la lista y, si son iguales, se devuelve la posición media.

```
if lista[medio] == nombre_buscado:
            return medio
```

Si el valor buscado precede al del medio, el algoritmo descarta todos los valores subsiguientes.

```
elif lista[medio] > nombre_buscado:
            fim = medio-1
```

Y si el valor buscado está después del valor medio, el algoritmo descarta todos los valores anteriores, hasta que solo quede el elemento deseado. Si el elemento restante no es lo que queremos, se devuelve un valor negativo.

```
elif lista[medio] < nombre_buscado:
            inicio=medio+1
```

Ahora bien, al realizar la simulación de las 3500 solicitudes, nos damos cuenta que la ejecución es casi instantánea. ¿Y por qué este algoritmo funciona tan rápido? ¿Calculamos?

Comparación de la eficiencia de los algoritmos

Considerando, una vez más, que tenemos 85.000 alumnos y que en cada iteración de búsqueda se descarta la mitad de la lista que no es la que buscamos, podemos calcular mediante logaritmos en base 2 y concluir que con cada petición del función de búsqueda,

en el peor de los casos, se realizan operaciones lg(N), es decir, lg(85000) ≃ 16 operaciones.

Pero aún no ha terminado, ya que hicimos 3500 llamadas a esta función, por lo que realizamos (3500 * 16) ≃ 56000 operaciones.

Así, pudimos optimizar nuestro algoritmo que antes realizaba casi 300 millones de operaciones a solo 56 mil. ¡De esta manera, podemos ir más seguros para la campaña que traerá 3000 estudiantes más a nuestra plataforma!

Una vez más se mostró la importancia de pensar en cómo se comportarán nuestros algoritmos de acuerdo a la cantidad de datos recibidos, y vale la pena recalcar que aunque existan abstracciones de tales funciones, como la función index() que realizaría esta búsqueda con solo una llamada, es muy importante aprender estos fundamentos. ¿Pero por qué? ¿Dónde más podemos aplicar esto?

> *Vea que para la solución de búsqueda binaria es necesario tener la lista ordenada, por lo que usamos la función `sorted()` para realizar esta tarea. ¿Pero imagina que el lenguaje no nos ofreciera tal función? ¿Como podríamos hacerlo? Cubriremos este tema en el artículo de clasificación de Python, ¡nos vemos allí!*

Brendo Rodrigo Souza de Matos

Ingeniero de Software apasionado por lo que hace, amante de los nuevos retos y sediento de conocimiento. Actualmente soy Ingeniero de Software de Plataformas en Méliuz (B3: CASH3) y estoy realizando una Maestría en Ciencias de la Computación en la Universidad Federal de Amazonas.

Este articulo fue adecuado para Alura Latam por: Wilfredo Rojas

Lógica de Programación | Java

Cursos de Front End

HTML y CSS | JavaScript | React

Cursos de Data Science

Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

Cursos de DevOps

Docker | Linux

Cursos de Innovación y Gestión

Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento